Exhibit 3

U.S. Patent No. 8,190,610
Charles Schwab Corporation

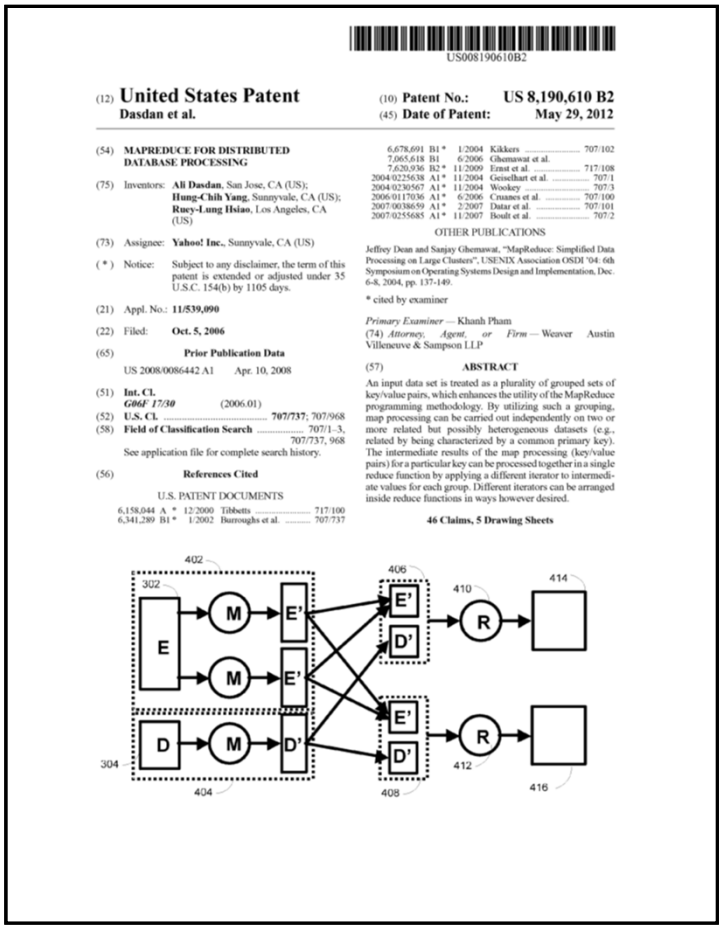
nba nelson bumgardner albritton

Claim Chart for Representative Claim 1

OVERVIEW

nelson bumgardner albritton

U.S. Patent No. 8,190,610



US008190610B2

(12) **United States Patent**
Dasdan et al.

(10) **Patent No.: US 8,190,610 B2**
(45) **Date of Patent: May 29, 2012**

(54) **MAPREDUCE FOR DISTRIBUTED DATABASE PROCESSING**

(75) Inventors: **Ali Dasdan**, San Jose, CA (US); **Hung-Chih Yang**, Sunnyvale, CA (US); **Ruey-Lung Hsiao**, Los Angeles, CA (US)

(73) Assignee: **Yahoo! Inc.**, Sunnyvale, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 1105 days.

(21) Appl. No.: **11/539,090**

(22) Filed: **Oct. 5, 2006**

(65) **Prior Publication Data**

US 2008/0086442 A1 Apr. 10, 2008

(51) **Int. Cl.**
G06F 17/30 (2006.01)

(52) **U.S. Cl.** **707/737**; 707/968

(58) **Field of Classification Search** 707/1–3, 707/737, 968

See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

6,158,044 A * 12/2000 Tibbetts 717/100
6,341,289 B1 * 1/2002 Burroughs et al. 707/737
6,678,691 B1 * 1/2004 Kikkers 707/102
7,065,618 B1 6/2006 Ghemawat et al.
7,620,936 B2 * 11/2009 Ernst et al. 717/108
2004/0225638 A1 * 11/2004 Geiselhart et al. 707/1
2004/0230567 A1 * 11/2004 Wookey 707/3
2006/0117036 A1 * 6/2006 Cruanes et al. 707/100
2007/0038659 A1 * 2/2007 Datar et al. 707/101
2007/0255685 A1 * 11/2007 Boult et al. 707/2

OTHER PUBLICATIONS

Jeffrey Dean and Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", USENIX Association OSDI '04: 6th Symposium on Operating Systems Design and Implementation, Dec. 6-8, 2004, pp. 137-149.

* cited by examiner

Primary Examiner — Khanh Pham
(74) *Attorney, Agent, or Firm* — Weaver Austin Villeneuve & Sampson LLP

(57) **ABSTRACT**

An input data set is treated as a plurality of grouped sets of key/value pairs, which enhances the utility of the MapReduce programming methodology. By utilizing such a grouping, map processing can be carried out independently on two or more related but possibly heterogeneous datasets (e.g., related by being characterized by a common primary key). The intermediate results of the map processing (key/value pairs) for a particular key can be processed together in a single reduce function by applying a different iterator to intermediate values for each group. Different iterators can be arranged inside reduce functions in ways however desired.

46 Claims, 5 Drawing Sheets

Title:	**MAPREDUCE FOR DISTRIBUTED DATABASE PROCESSING**
Priority Date:	**October 5, 2006**
Filing Date:	**October 5, 2006**
Issue Date:	**May 29, 2012**
Expiration Date:	**October 14, 2029**

(57) **ABSTRACT**

An input data set is treated as a plurality of grouped sets of key/value pairs, which enhances the utility of the MapReduce programming methodology. By utilizing such a grouping, map processing can be carried out independently on two or more related but possibly heterogeneous datasets (e.g., related by being characterized by a common primary key). The intermediate results of the map processing (key/value pairs) for a particular key can be processed together in a single reduce function by applying a different iterator to intermediate values for each group. Different iterators can be arranged inside reduce functions in ways however desired.

Representative Claim 1

A method of processing data of a data set over a distributed system, wherein the data set comprises a plurality of data groups, the method comprising:

partitioning the data of each one of the data groups into a plurality of data partitions that each have a plurality of key-value pairs and providing each data partition to a selected one of a plurality of mapping functions that are each user-configurable to independently output a plurality of lists of values for each of a set of keys found in such map function's corresponding data partition to form corresponding intermediate data for that data group and identifiable to that data group, wherein the data of a first data group has a different schema than the data of a second data group and the data of the first data group is mapped differently than the data of the second data group so that different lists of values are output for the corresponding different intermediate data, wherein the different schema and corresponding different intermediate data have a key in common; and

reducing the intermediate data for the data groups to at least one output data group, including processing the intermediate data for each data group in a manner that is defined to correspond to that data group, so as to result in a merging of the corresponding different intermediate data based on the key in common,

wherein the mapping and reducing operations are performed by a distributed system.

CLAIM CHART

nelson bumgardner albritton

References Cited

Upon information and belief, Charles Schwab has and continues to use a system called Big Data Analytics (hereinafter, the "Charles Schwab system"), which is an analytics platform built on technology from Datameer layered on top of an Apache Hadoop framework that includes the elements in the systems described in and/or practices the steps of the methods described in the claims of U.S. Patent No. 8,190,610 (the "'610 Patent").

The following chart presents R2 Solutions' analysis of the Charles Schwab system based on publicly available information.

The citations in the chart refer to the following publicly available documents, which are incorporated by reference as if fully set forth herein:

[1] Sachin P. Bappalige, "An introduction to Apache Hadoop for big data," opensource.com (Aug. 26, 2014), *available at* https://opensource.com/life/14/8/intro-apache-hadoop-big-data ("**An introduction to Apache Hadoop for big data**")

[2] MapReduce Tutorial, *available at* http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client- core/MapReduceTutorial.html ("**MapReduce Tutorial**")

[3] Interface Reducer<K2,V2,K3,V3>, *available at* https://hadoop.apache.org/docs/r2.8.0/api/org/apache/hadoop/mapred/Reducer.html ("**Interface Reducer**")

[4] ASF Infrabot, "HadoopMapReduce," Dashboard (July 9, 2019), *available at* https://cwiki.apache.org/confluence/display/HADOOP2/HadoopMapReduce ("**HadoopMapReduce**")

[5] Tom Davenport, "Feeding a Data-Hungry Organization at Charles Schwab," Forbes (Jun. 5, 2018), *available at* https://www.forbes.com/sites/tomdavenport/2018/06/05/feeding-a-data-hungry-organization-at-charles-schwab/?sh=541cbb3977bb ("**Feeding a Data-Hungry Organization**")

[6] "Charles Schwab Careers: Senior Data Engineer," *available at* https://career-schwab.icims.com/jobs/64871/senior-data-engineer---etl-lead/job?mobile=false&width=1230&height=500&bga=true&needsRedirect=false&jan1offset=-360&jun1offset=-300"

References Cited (continued)

[7] "*Charles Schwab Careers: Senior Software Engineer*," *available at* https://jobs.schwabjobs.com/job/san-francisco/senior-software-engineer/33727/17647572

[8] "*Internship at Charles Schwab – Summer 2019*," UT Dallas, *available at* https://jindal.utdallas.edu/som/internship-stories/1087

[9] *LinkedIn Profile: Guarav Singh*, *available at* https://www.linkedin.com/in/gaurav-singh-82812614 ("**LinkedIn Profile**")

[10] Hive 2.1.1 API – Class Schema, *available at* https://hive.apache.org/javadocs/r2.1.1/api/org/apache/hadoop/hive/metastore/api/Schema.html ("**Class Schema**")

[11] Confluence Administrator, "Hive Tutorial," Dashboard (updated by Owen O'Malley, Mar. 5, 2019), *available at* https://cwiki.apache.org/confluence/display/Hive/Tutorial ("**Hive Tutorial**")

[12] Confluence Administrator, "LanguageManual Joins," Dashboard (updated by Dudu Markovitz, June 5, 2017), *available at* https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Joins ("**LanguageManual Joins**")

[13] Tanel Poder, "Speed Up Your Queries with Hive LLAP Engine on Hadoop or in the Cloud," Gluent, *available at* https://www.slideshare.net/gluent/speed-up-your-queries-with-hive-llap-engine-on-hadoop-or-in-the-cloud ("**Speed Up Your Queries with Hive LLAP Engine**")

[14] "Apache Hive," Wikipedia, *available at* https://en.wikipedia.org/wiki/Apache_Hive ("**Apache Hive**")

[15] Petar Zecevic & Marko Bonaci, "Spark in Action" (Manning Publications Nov. 2016), *available at* https://learning.oreilly.com/library/view/spark-in-action/9781617292606/kindle_split_014.html ("**Spark in Action**")

[16] "Apache Spark FAQ," *available at* https://spark.apache.org/faq.html ("**Apache Spark FAQ**")

References Cited (continued)

[17] Apache Spark 3.0.1, "RDD Programming Guide," *available at* http://spark.apache.org/docs/latest/rdd-programming-guide.html ("**RDD Programming Guide**")

[18] "Learning Spark," O'Reilly Media, Inc. (February 2015), *available at* https://learning.oreilly.com/library/view/learning-spark/9781449359034/ch04.html#chap-pair-RDDS ("**Learning Spark**")

[19] "Spark RDD Transformations with examples," *available at* https://sparkbyexamples.com/apache-spark-rdd/spark-rdd-transformations/ ("**Spark RDD Transformations**")

[20] Shrey Mehrotra & Akash Grade, "Apache Quick Start Guide" (Packt Publishing, Jan. 2019), *available at* https://learning.oreilly.com/library/view/apache-spark-quick/9781789349108/0ee1a5e2-09d0-49f0-99f5-9dee8336258d.xhtml ("**Apache Spark Quick Start Guide**")

[21] "Spark SQL Shuffle Partitions," *available at* https://sparkbyexamples.com/spark/spark-shuffle-partitions/ ("**Spark SQL Shuffle Partitions**")

[22] "Apache Spark Map vs. FlatMap Operation," DataFlair, *available at* https://data-flair.training/blogs/apache-spark-map-vs-flatmap/ ("**Apache Spark Map vs. FlatMap**")

[23] Confluence Administrator, "LanguageManual Transform,"Dashboard (updated by Sushanth Sowmyan on Dec. 9, 2015), *available at* https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Transform ("**LanguageManual Transform**")

Claim Chart – Claim 1

Claim Feature	Evidence from Charles Schwab
A method of processing data of a data set over a distributed system, wherein the data set comprises a plurality of data groups, the method comprising:	Charles Schwab infringes Claim 1 of the '610 Patent. The Charles Schwab system performs a method of processing data of a data set over a distributed system, wherein the data set comprises a plurality of data groups, the method comprising the steps discussed below: Charles Schwab uses an Apache Hadoop distributed file system that relies on MapReduce. **[5] Feeding a Data-Hungry Organization:** At about the same time, Schwab felt that it needed to modernize its data environment and Andrew Salesky was named head of Global Data. The Global Data Technology organization was also formed, reporting to the CIO. This led to incorporating a Hadoop-based big data lake and other new technologies. Salesky notes that Schwab, like any other organization wanting to do more with advanced analytics and AI, needed to make some "long tailed" investments in its data. In terms of data integration and master data management, the primary focus at Schwab has been on client data and developing consistent enterprise definitions of key data assets.

Claim Chart – Claim 1

Claim Feature	Evidence from Charles Schwab
A method of processing data of a data set over a distributed system, wherein the data set comprises a plurality of data groups, the method comprising:	Upon information and belief, the Charles Schwab System uses Apache Hadoop, Hive, and Spark: **[6] Charles Schwab Careers: Senior Data Engineer:** **Your Opportunity** Charles Schwab & Co., Inc is currently seeking a seasoned ETL Lead with a passion for hands-on design/development and collaboration with our business partners. The ideal candidate is adept at using large data sets to find opportunities for product and process optimization to test the effectiveness of different courses of action. The ETL Lead must have deep experience in Enterprise Data Warehouse, Data Mart design, development and end to end execution of data solutions. Prior experience in developing design patterns for Big Data and Cloud solutions is a big plus. This role will require hands on development with wide range of technical skills which include data analysis, ETL (Talend / Informatica, Unix, Big Data Technologies) and modeling skills (SQL, ERwin). **What you are good at** This position is part of the Global Data Technology (GDT) organization that governs the strategy and implementation of the enterprise data warehouse and emerging data platforms. The ETL Lead will be designing, building and supporting data processing pipelines to transform data using Hadoop technologies. You'll be designing schemas, data models and data architecture for Hadoop and HBase environments. You'll be implementing data flow scripts using Unix / Hive QL / Pig scripting. You'll be designing, building data assets in MapR-DB (HBASE), and HIVE. You'll be developing and executing quality assurance and test scripts. You'll be work with product owners and business analysts to understand business requirements and use cases to design solutions. You'll have the opportunity to grow in responsibility, work on exciting and challenging projects, train on emerging technologies and help set the future of the Data Solution Delivery team. You'll lead investigation and resolution efforts for critical/high impact problems, defects and incidents. Provides technical guidance to team members.

Claim Chart – Claim 1

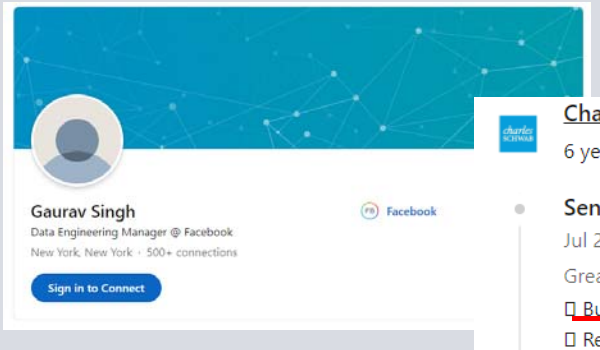
Claim Feature	Evidence from Charles Schwab
A method of processing data of a data set over a distributed system, wherein the data set comprises a plurality of data groups, the method comprising:	Upon information and belief, the Charles Schwab System uses Apache Hadoop, Hive, and Spark: **[7] Charles Schwab Careers: Senior Software Engineer:** **Responsibilities** • Build reliable data pipelines using Python and Scala (with Apache Spark) to ingest, process and transform data • Enable easy interaction with processed data by exposing REST-like endpoints • Write algorithms that interact with the processed data to transform datasets into portfolios based on various parameters • Add functionality to a web application to wrap the backend services and make them easier to use • Write extensive and reliable tests at all parts of the test pyramid • Continuously make improvements and optimizations so that we can build portfolios using large amounts of data in seconds • Write automated tests for all code to ensure the highest quality product • Contribute to architectural decisions related to the platform • Build infrastructure required to deploy the code in the cloud using Terraform

Claim Chart – Claim 1

Claim Feature	Evidence from Charles Schwab
A method of processing data of a data set over a distributed system, wherein the data set comprises a plurality of data groups, the method comprising:	Upon information and belief, the Charles Schwab System uses Apache Hadoop, Hive, and Spark: **[8] Internship at Charles Schwab – Summer 2019:**  I am involved in the 2 major projects and a small project. The major projects I dealt with were connected to the Digital Accelerators namely DARE and ONCE. In this project I had to make and automate the Web Analytics DARE and ONCE dashboards in Tableau. Initially the web dashboards used to get updated with the manual intervention by pulling out the Adobe Analytics Data and storing it into excel and then connecting Excel to Tableau. For automating this process, the Adobe Analytics Data was loaded into the Big Data Environment where the system for the project was Hadoop and I used the Nexus Query Chameleon to query the big data with the help of Apache Hive – HQL. Results of the queries were stored in the IDW database. There after the database was connected to tableau and I prepared the Dashboards. Automation of this whole process is done by running some of the UNIX jobs.

Claim Chart – Claim 1

Claim Feature	Evidence from Charles Schwab
A method of processing data of a data set over a distributed system, wherein the data set comprises a plurality of data groups, the method comprising:	Upon information and belief, the Charles Schwab System uses Apache Hadoop, Hive, and Spark: **[9] LinkedIn Profile:**



Charles Schwab

6 years 2 months

Senior Manager, Data Engineering

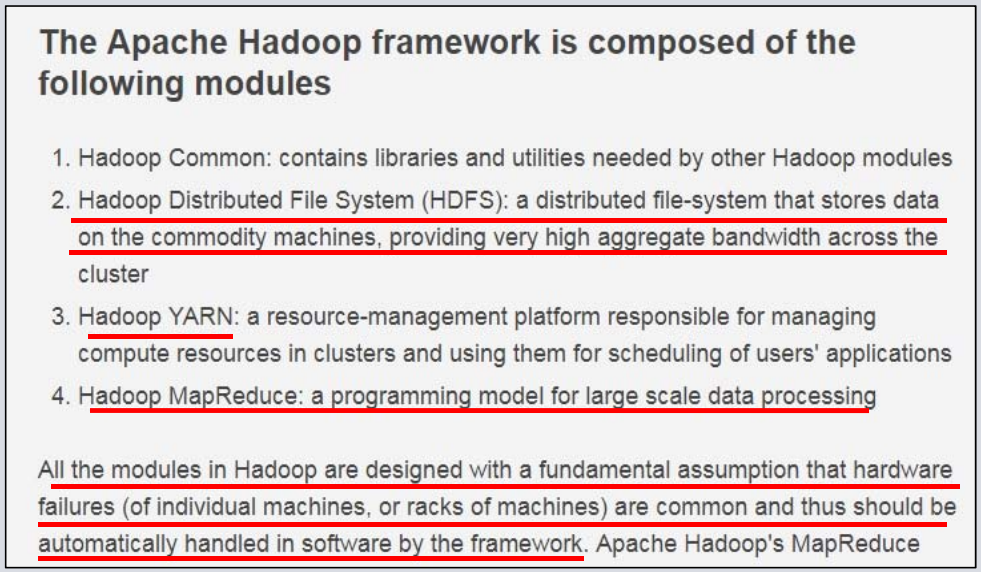
Jul 2016 - Jul 2019 · 3 years 1 month

Greater Denver Area

- Building strong data pipeline using HDFS, Python, Hive, Spark, Kafka & Teradata.
- Responsible for the development of a big data customer behavior hub to enable cross channel attribution and marketing optimization using HDFS, Hive, SPARK & Python.
- Responsible for Strategy, Design & Delivery functions aimed at creating a robust analytical data layer on Enterprise Data warehouse platform to help facilitate "one version of the truth" reporting across multiple product and delivery
- Manage a team 35+ engineers in diverse geographical locations and varied roles like Data Engineer, Business System Analyst, Scrum masters etc.
- Responsible for multi million project portfolio annually.

Show less

Claim Chart – Claim 1

Claim Feature	Evidence from Charles Schwab
A method of **processing data of a data set over a distributed system**, wherein the data set comprises a plurality of data groups, the method comprising:	Charles Schwab's Apache Hadoop MapReduce implementation is over a distributed system: **[1] An introduction to Apache Hadoop for big data at 2:** 

The Apache Hadoop framework is composed of the following modules

1. Hadoop Common: contains libraries and utilities needed by other Hadoop modules
2. Hadoop Distributed File System (HDFS): a distributed file-system that stores data on the commodity machines, providing very high aggregate bandwidth across the cluster
3. Hadoop YARN: a resource-management platform responsible for managing compute resources in clusters and using them for scheduling of users' applications
4. Hadoop MapReduce: a programming model for large scale data processing

All the modules in Hadoop are designed with a fundamental assumption that hardware failures (of individual machines, or racks of machines) are common and thus should be automatically handled in software by the framework. Apache Hadoop's MapReduce

Claim Chart – Claim 1

Claim Feature	Evidence from Charles Schwab
A method of **processing data of a data set over a distributed system**, wherein the data set comprises a plurality of data groups, the method comprising:	**[14]** **Apache Hive:** **Apache Hive** is a data warehouse software project built on top of Apache Hadoop for providing data query and analysis.[3] Hive gives an SQL-like interface to query data stored in various databases and file systems that integrate with Hadoop. Traditional SQL queries must be implemented in the MapReduce Java API to execute SQL applications and queries over distributed data. Hive provides the necessary SQL abstraction to integrate SQL-like queries (HiveQL) into the underlying Java without the need to implement queries in the low-level Java API. Since most data warehousing applications work with SQL-based querying languages, Hive aids portability of SQL-based applications to Hadoop.[4] While initially developed by Facebook, Apache Hive is used and developed by other companies such as Netflix and the Financial Industry Regulatory Authority (FINRA).[5][6] Amazon maintains a software fork of Apache Hive included in Amazon Elastic MapReduce on Amazon Web Services.[7]

Claim Chart – Claim 1

Claim Feature	Evidence from Charles Schwab
A method of processing data of a data set over a distributed system, wherein the data set comprises a plurality of data groups, the method comprising:	**[16] Apache Spark FAQ at 1:** How does Spark relate to Apache Hadoop? Spark is a fast and general processing engine compatible with Hadoop data. It can run in Hadoop clusters through YARN or Spark's standalone mode, and it can process data in HDFS, HBase, Cassandra, Hive, and any Hadoop InputFormat. It is designed to perform both batch processing (similar to MapReduce) and new workloads like streaming, interactive queries, and machine learning. **[17] RDD Programming Guide at 4, 2:** Resilient Distributed Datasets (RDDs) Spark revolves around the concept of a *resilient distributed dataset* (RDD), which is a fault-tolerant collection of elements that can be operated on in parallel. There are two ways to create RDDs: *parallelizing* an existing collection in your driver program, or referencing a dataset in an external storage system, such as a shared filesystem, HDFS, HBase, or any data source offering a Hadoop InputFormat. Overview At a high level, every Spark application consists of a *driver program* that runs the user's `main` function and executes various *parallel operations* on a cluster. The main abstraction Spark provides is a *resilient distributed dataset* (RDD), which is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel. RDDs are created by starting with a file in the Hadoop file system (or any other Hadoop-supported file system), or an existing Scala collection in the driver program, and transforming it. Users may also ask Spark to *persist* an RDD in memory, allowing it to be reused efficiently across parallel operations. Finally, RDDs automatically recover from node failures.

Claim Chart – Claim 1

Claim Feature	Evidence from Charles Schwab
partitioning the data of each one of the data groups into a **plurality of data partitions** that each have a plurality of **key-value pairs** and providing each **data partition** to a selected one of a plurality of **mapping functions that are each user-configurable** to independently output a plurality of lists of values for each of a set of keys found in such map function's corresponding data partition to form corresponding intermediate data for that data group and identifiable to that data group,	The method performed by the Charles Schwab system includes partitioning the data of each one of the data groups into a plurality of data partitions that each have a plurality of key-value pairs and providing each data partition to a selected one of a plurality of mapping functions that are each user-configurable to independently output a plurality of lists of values for each of a set of keys found in such map function's corresponding data partition to form corresponding intermediate data for that data group and identifiable to that data group. In Charles Schwab's Hadoop MapReduce framework, the input data is partitioned into independent chunks (data partitions), which are processed by Mapper functions: **[2] MapReduce Tutorial at 2:** A MapReduce *job* usually splits the input data-set into independent chunks which are processed by the *map tasks* in a completely parallel manner. The framework sorts the outputs of the maps, which are then input to the *reduce tasks*. Typically both the input and the output of the job are stored in a file-system. The framework takes care of scheduling tasks, monitoring them and re-executes the failed tasks.

Claim Chart – Claim 1

Claim Feature	Evidence from Charles Schwab
partitioning the data of each one of the data groups into a **plurality of data partitions** that each have a plurality of **key-value pairs** and providing each **data partition** to a selected one of a plurality of **mapping functions that are each user-configurable** to independently output a plurality of lists of values for each of a set of keys found in such map function's corresponding data partition to form corresponding intermediate data for that data group and identifiable to that data group,	The data partitions with key-value pairs are passed to user-configurable Mapper functions for further processing: **[4] HadoopMapReduce at 1:** As key-value pairs are read from the RecordReader they are passed to the configured Mapper. The user supplied Mapper does whatever it wants with the input pair and calls OutputCollector.collect with key-value pairs of its own choosing. The output it generates must use one key class and one value class. This is because the Map output will be written into a SequenceFile which has per-file type information and all the records must have the same type (use subclassing if you want to output different data structures). The Map input and output key-value pairs are not necessarily related typewise or in cardinality.

Claim Chart – Claim 1

Claim Feature	Evidence from Charles Schwab
partitioning the data of each one of the data groups into a plurality of data partitions that each have a plurality of key-value pairs and providing each data partition to a selected one of a plurality of **mapping functions that are each user-configurable** to independently output a plurality of lists of values for each of a set of keys found in such **map function**'s corresponding data partition to form corresponding **intermediate data** for that data group and identifiable to that data group,	The Mapper function maps the key-value pairs to form intermediate key-value pairs: **[2] MapReduce Tutorial at 7:** **Mapper** Mapper maps input key/value pairs to a set of intermediate key/value pairs. Maps are the individual tasks that transform input records into intermediate records. The transformed intermediate records do not need to be of the same type as the input records. A given input pair may map to zero or many output pairs.

Claim Chart – Claim 1

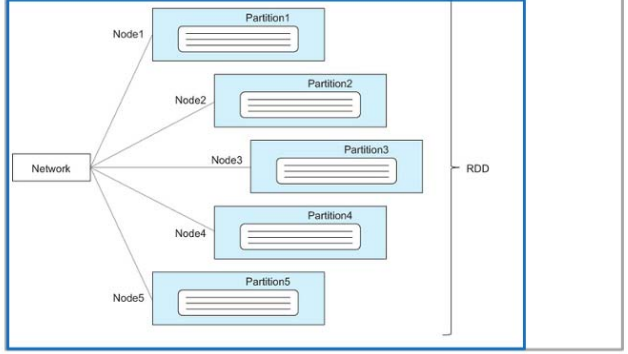
Claim Feature	Evidence from Charles Schwab
partitioning the data of each one of the data groups into a plurality of data partitions that each have a plurality of key-value pairs and providing each data partition to a selected one of a plurality of mapping functions that are each user-configurable to independently output a plurality of lists of values for each of a set of keys found in such map function's corresponding data partition to form corresponding intermediate data for that data group and identifiable to that data group,	With Spark, a data set is first partitioned into a plurality of elements and distributed across nodes. This distribution of elements is called a resilient distributed dataset (RDD): **[17] RDD Programming Guide at 2, 4, 5:** **Overview** At a high level, every Spark application consists of a *driver program* that runs the user's `main` function and executes various *parallel operations* on a cluster. The main abstraction Spark provides is a *resilient distributed dataset* (RDD), which is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel. RDDs are created by starting with a file in the Hadoop file system (or any other Hadoop-supported file system), or an existing Scala collection in the driver program, and transforming it. Users may also ask Spark to *persist* an RDD in memory, allowing it to be reused efficiently across parallel operations. Finally, RDDs automatically recover from node failures. **Resilient Distributed Datasets (RDDs)** Spark revolves around the concept of a *resilient distributed dataset* (RDD), which is a fault-tolerant collection of elements that can be operated on in parallel. There are two ways to create RDDs: *parallelizing* an existing collection in your driver program, or referencing a dataset in an external storage system, such as a shared filesystem, HDFS, HBase, or any data source offering a Hadoop InputFormat. One important parameter for parallel collections is the number of *partitions* to cut the dataset into. Spark will run one task for each partition of the cluster. Typically you want 2-4 partitions for each CPU in your cluster. Normally, Spark tries to set the number of partitions automatically based on your cluster. However, you can also set it manually by passing it as a second parameter to `parallelize` (e.g. `sc.parallelize(data, 10)`). Note: some places in the code use the term slices (a synonym for partitions) to maintain backward compatibility.

Claim Chart – Claim 1

Claim Feature	Evidence from Charles Schwab
partitioning the data of each one of the data groups into a plurality of data partitions that each have a plurality of key-value pairs and providing each data partition to a selected one of a plurality of mapping functions that are each user-configurable to independently output a plurality of lists of values for each of a set of keys found in such map function's corresponding data partition to form corresponding intermediate data for that data group and identifiable to that data group,	**[15]** **Spark in Action:**

Each part (piece or slice) of an RDD is called a *partition*.[2] When you load a text file from your local filesystem into Spark, for example, the file's contents are split into partitions, which are evenly distributed to nodes in a cluster. More than one partition may end up on the same node. The sum of all those partitions forms your RDD. This is where the word *distributed* in *resilient distributed dataset* comes from. Figure 4.1 shows the distribution of lines of a text file loaded into an RDD in a five-node cluster. The original file had 15 lines of text, so each RDD partition was formed with 3 lines of text. Each RDD maintains a list of its partitions and an optional list of preferred locations for computing the partitions.

Figure 4.1. Simplified look at partitions of an RDD in a five-node cluster. The RDD was created by loading a text file using the `textFile` method of `SparkContext`. The loaded text file had 15 lines of text, so each partition was formed with 3 lines of text.



Claim Chart – Claim 1

Claim Feature	Evidence from Deezer
partitioning the data of each one of the data groups into a plurality of **data partitions that each have a plurality of key-value pairs** and providing each data partition to a selected one of a plurality of mapping functions that are each user-configurable to independently output a plurality of lists of values for each of a set of keys found in such map function's corresponding data partition to form corresponding intermediate data for that data group and identifiable to that data group,	One form of the RDDs is a pair RDD, which includes key/value pairs: **[18] Learning Spark:** Spark provides special operations on RDDs containing key/value pairs. These RDDs are called pair RDDs. Pair RDDs are a useful building block in many programs, as they expose operations that allow you to act on each key in parallel or regroup data across the network. For example, pair RDDs have a `reduceByKey()` method that can aggregate data separately for each key, and a `join()` method that can merge two RDDs together by grouping elements with the same key. It is common to extract fields from an RDD (representing, for instance, an event time, customer ID, or other identifier) and use those fields as keys in pair RDD operations. **[15] Spark in Action:** **4.1. WORKING WITH PAIR RDDS** Storing data as key-value pairs offers a simple, general, and extensible data model because each key-value pair can be stored independently and it's easy to add new types of keys and new types of values. This extensibility and simplicity have made this practice fundamental to several frameworks and applications. For example, many popular caching systems and NoSQL databases, such as memcached and Redis, are key-value stores. Hadoop's MapReduce also operates on key-value pairs (as you can see in appendix A). Keys and values can be simple types such as integers and strings, as well as more-complex data structures. Data structures traditionally used to represent key-value pairs are *associative arrays*, also called *dictionaries* in Python and *maps* in Scala and Java. In Spark, RDDs containing key-value tuples are called *pair RDDs*. Although you don't have to use data in Spark in the form of key-value pairs (as you've seen in the previous chapters), pair RDDs are well suited (and indispensable) for many use cases. Having keys along with the data enables you to aggregate, sort, and join the data, as you'll soon see. But before doing any of that, the first step is to create a pair RDD, of course.

Claim Chart – Claim 1

Claim Feature	Evidence from Charles Schwab
partitioning the data of each one of the data groups into a plurality of data partitions that each have a plurality of key-value pairs and **providing each data partition to a selected one of a plurality of mapping functions** that are each user-configurable to independently output a plurality of lists of values for each of a set of keys found in such map function's corresponding data partition to form corresponding intermediate data for that data group and identifiable to that data group,	The RDDs support two types of data operations: transformations and actions. **[17] RDD Programming Guide at 6:** **RDD Operations** RDDs support two types of operations: *transformations*, which create a new dataset from an existing one, and *actions*, which return a value to the driver program after running a computation on the dataset. For example, `map` is a transformation that passes each dataset element through a function and returns a new RDD representing the results. On the other hand, `reduce` is an action that aggregates all the elements of the RDD using some function and returns the final result to the driver program (although there is also a parallel `reduceByKey` that returns a distributed dataset). All transformations in Spark are *lazy*, in that they do not compute their results right away. Instead, they just remember the transformations applied to some base dataset (e.g. a file). The transformations are only computed when an action requires a result to be returned to the driver program. This design enables Spark to run more efficiently. For example, we can realize that a dataset created through `map` will be used in a `reduce` and return only the result of the `reduce` to the driver, rather than the larger mapped dataset.

Claim Chart – Claim 1

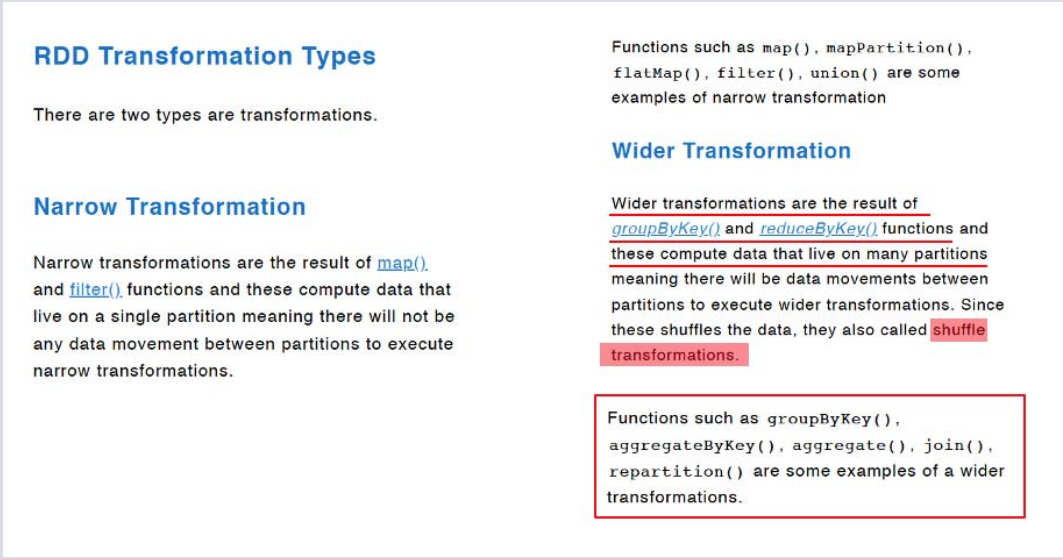
Claim Feature	Evidence from Charles Schwab
partitioning the data of each one of the data groups into a plurality of data partitions that each have a plurality of key-value pairs and **providing each data partition to a selected one of a plurality of mapping functions** that are each user-configurable to independently output a plurality of lists of values for each of a set of keys found in such map function's corresponding data partition to form corresponding intermediate data for that data group and identifiable to that data group,	Examples of transformation functions: **[17] RDD Programming Guide at 11–12:**

Transformations

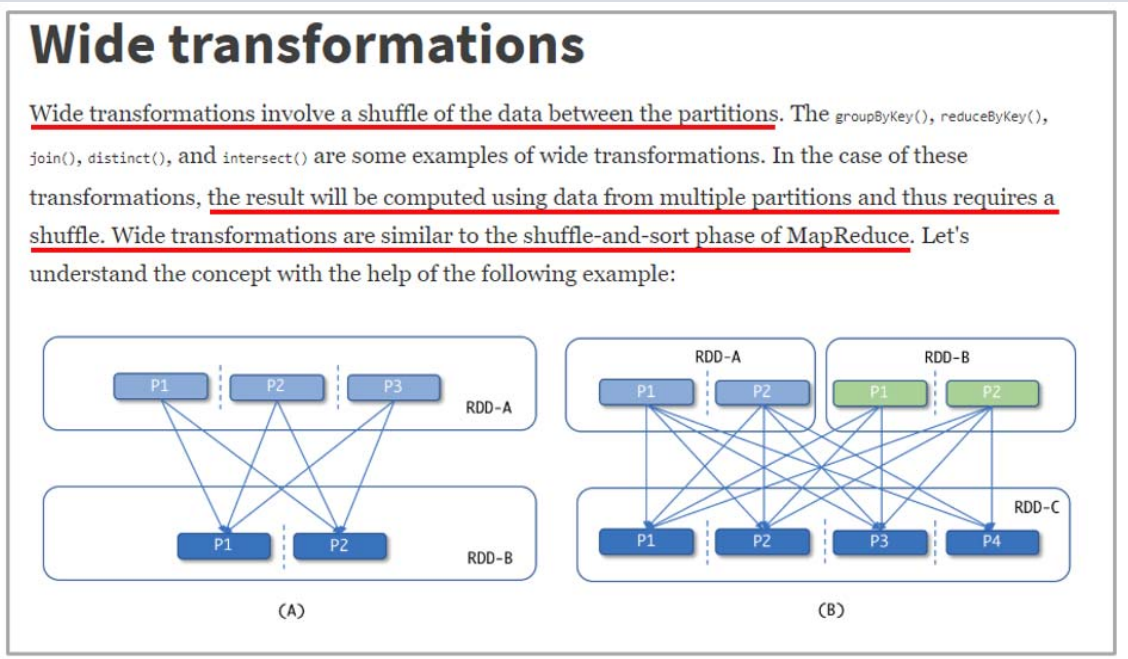
The following table lists some of the common transformations supported by Spark. Refer to the RDD API doc (Scala, Java, Python, R) and pair RDD functions doc (Scala, Java) for details.

Transformation	Meaning
map(*func*)	Return a new distributed dataset formed by passing each element of the source through a function *func*.
filter(*func*)	Return a new dataset formed by selecting those elements of the source on which *func* returns true.
flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).
mapPartitions(*func*)	Similar to map, but runs separately on each partition (block) of the RDD, so *func* must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.
mapPartitionsWithIndex(*func*)	Similar to mapPartitions, but also provides *func* with an integer value representing the index of the partition, so *func* must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.
groupByKey([*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. **Note:** If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using `reduceByKey` or `aggregateByKey` will yield much better performance. **Note:** By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional `numPartitions` argument to set a different number of tasks.
reduceByKey(*func*, [*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) => V. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.
aggregateByKey(*zeroValue*)(*seqOp*, *combOp*, [*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.

Claim Chart – Claim 1

Claim Feature	Evidence from Charles Schwab
partitioning the data of each one of the data groups into a plurality of data partitions that each have a plurality of key-value pairs and **providing each data partition to a selected one of a plurality of mapping functions** that are each user-configurable to independently output a plurality of lists of values for each of a set of keys found in such map function's corresponding data partition to form corresponding intermediate data for that data group and identifiable to that data group,	The transformation operations include narrow and wide transformations. Wide transformations (also called "shuffle") operate on data from multiple partitions. Wide transformations include aggregateByKey, reduceByKey, etc. **[19] Spark RDD Transformations at 2–3:**  **RDD Transformation Types** There are two types are transformations. **Narrow Transformation** Narrow transformations are the result of map() and filter() functions and these compute data that live on a single partition meaning there will not be any data movement between partitions to execute narrow transformations. Functions such as `map()`, `mapPartition()`, `flatMap()`, `filter()`, `union()` are some examples of narrow transformation **Wider Transformation** Wider transformations are the result of *groupByKey()* and *reduceByKey()* functions and these compute data that live on many partitions meaning there will be data movements between partitions to execute wider transformations. Since these shuffles the data, they also called shuffle transformations. Functions such as `groupByKey()`, `aggregateByKey()`, `aggregate()`, `join()`, `repartition()` are some examples of a wider transformations.

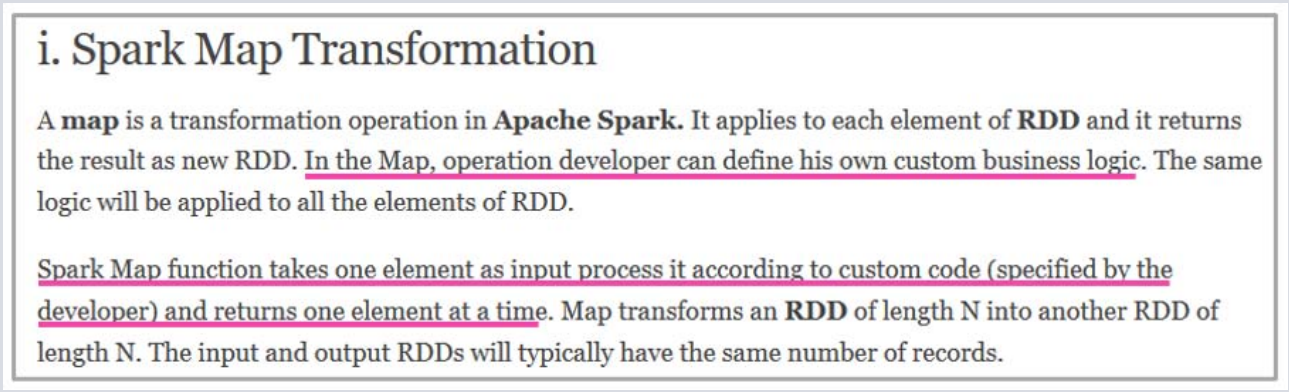
Claim Chart – Claim 1

Claim Feature	Evidence from Charles Schwab
partitioning the data of each one of the data groups into a plurality of data partitions that each have a plurality of key-value pairs and **providing each data partition to a selected one of a plurality of mapping functions** that are each user-configurable to independently output a plurality of lists of values for each of a set of keys found in such map function's corresponding data partition to form corresponding intermediate data for that data group and identifiable to that data group,	**[20] Apache Spark Quick Start Guide:** 

Claim Chart – Claim 1

Claim Feature	Evidence from Charles Schwab
partitioning the data of each one of the data groups into a plurality of data partitions that each have a plurality of key-value pairs and **providing each data partition to a selected one of a plurality of mapping functions** that are each user-configurable to independently output a plurality of lists of values for each of a set of keys found in such map function's corresponding data partition to form corresponding intermediate data for that data group and identifiable to that data group,	When executing a shuffle transformation, a map task is first run on all data partitions. **[21] Spark SQL Shuffle Partitions:** **What is Spark Shuffle?** Shuffling is a mechanism Spark uses to redistribute the data across different executors and even across machines. Spark shuffling triggers when we perform certain transformation operations like `gropByKey()`, `reducebyKey()`, `join()` on RDD and DataFrame. When creating an RDD or DataFrame, Spark doesn't necessarily store the data for all keys in a partition since at the time of creation there is no way we can set the key for data set. Hence, when we run the reduceByKey() operation to aggregate the data on keys, Spark does the following. needs to first run tasks to collect all the data from all partitions and For example, when we perform reduceByKey() operation, Spark does the following • Spark first runs *map* tasks on all partitions which groups all values for a single key. • The results of the map tasks are kept in memory. • When results do not fit in memory, Spark stores the data into a disk. • Spark shuffles the mapped data across partitions, some times it also stores the shuffled data into a disk for reuse when it needs to recalculate. • Run the garbage collection • Finally runs reduce tasks on each partition based on key.

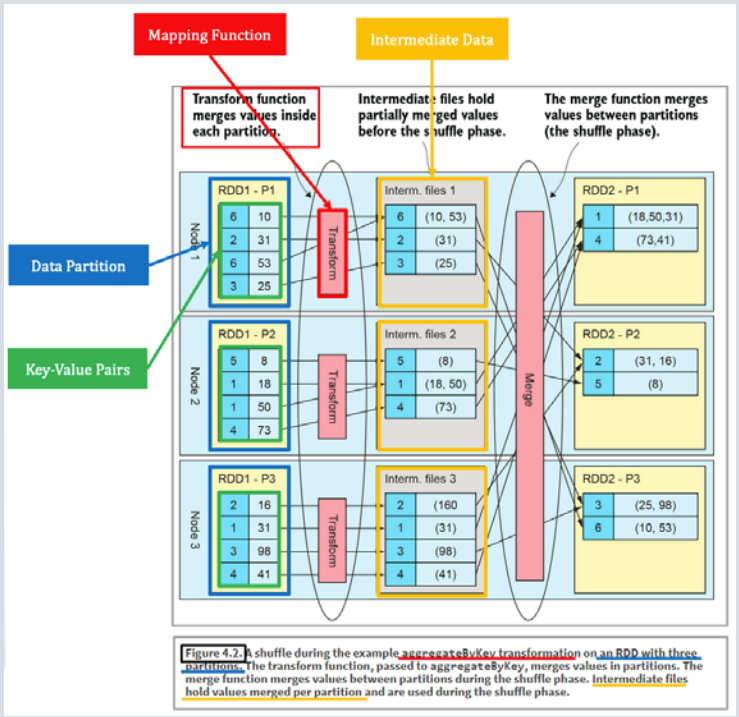
Claim Chart – Claim 1

Claim Feature	Evidence from Charles Schwab
partitioning the data of each one of the data groups into a plurality of data partitions that each have a plurality of key-value pairs and providing each data partition to a selected one of **a plurality of mapping functions that are each user-configurable** to independently output a plurality of lists of values for each of a set of keys found in such map function's corresponding data partition to form corresponding intermediate data for that data group and identifiable to that data group,	**[22] Apache Spark Map vs. FlatMap at 3–4:**  i. Spark Map Transformation A **map** is a transformation operation in **Apache Spark.** It applies to each element of **RDD** and it returns the result as new RDD. In the Map, operation developer can define his own custom business logic. The same logic will be applied to all the elements of RDD. Spark Map function takes one element as input process it according to custom code (specified by the developer) and returns one element at a time. Map transforms an **RDD** of length N into another RDD of length N. The input and output RDDs will typically have the same number of records.

Claim Chart – Claim 1

Claim Feature	Evidence from Charles Schwab
partitioning the data of each one of the data groups into a **plurality of data partitions that each have a plurality of key-value pairs** and **providing each data partition to a selected one of a plurality of mapping functions** that are each user-configurable **to independently output a plurality of lists of values for each of a set of keys found in such map function's corresponding data partition to form corresponding intermediate data for that data group and identifiable to that data group,**	When a mapping function is applied, intermediate files are created for that partition. **[15] Spark in Action:** **4.2.2. Understanding and avoiding unnecessary shuffling** Physical movement of data between partitions is called *shuffling*. It occurs when data from multiple partitions needs to be combined in order to build partitions for a new RDD. When grouping elements by key, for example, Spark needs to examine all of the RDD's partitions, find elements with the same key, and then physically group them, thus forming new partitions. To visualize what happens with partitions during a shuffle, we'll use the previous example with the `aggregateByKey` transformation (from section 4.1.2). The shuffle that occurs during that transformation is illustrated in figure 4.2. The transform function puts all values of each key in a single partition (partitions P1 to P3) into a list. Spark then writes this data to intermediate files on each node. In the next phase, the merge function is called to merge lists from different partitions, but of the same key, into a single list for each key. The default partitioner (`HashPartitioner`) then kicks in and puts each key in its proper partition. Tasks that immediately precede and follow the shuffle are called *map* and *reduce* tasks, respectively. The results of map tasks are written to intermediate files (often to the OS's filesystem cache only) and read by reduce tasks. In addition to being written to disk, the data is sent over the network, so it's important to try to minimize the number of shuffles during Spark jobs.

Claim Chart – Claim 1

Claim Feature	Evidence from Charles Schwab
partitioning the data of each one of the data groups into a **plurality of data partitions that each have a plurality of key-value pairs** and **providing each data partition to a selected one of a plurality of mapping functions** that are each user-configurable **to independently output a plurality of lists of values for each of a set of keys found in such map function's corresponding data partition to form corresponding intermediate data for that data group and identifiable to that data group,**	In this example, the aggregateByKey transformation operation (shuffle) is performed on three exemplary partitions: **[15] Spark in Action:** 

Claim Chart – Claim 1

Claim Feature	Evidence from Charles Schwab
wherein the data of a first data group has a different schema than the data of a second data group and **the data of the first data group is mapped differently than the data of the second data group** so that different lists of values are output for the corresponding different intermediate data, wherein the different schema and corresponding different intermediate data have a key in common; and	Users of the Charles Schwab System can create custom Mapper functions using Hive language and embed them into the MapReduce processing: **[11] Hive Tutorial:** **Custom Map/Reduce Scripts** Users can also plug in their own custom mappers and reducers in the data stream by using features natively supported in the Hive language. for example, in order to run a custom mapper script - map_script - and a custom reducer script - reduce_script - the user can issue the following command which uses the TRANSFORM clause to embed the mapper and the reducer scripts.

Claim Chart – Claim 1

Claim Feature	Evidence from Charles Schwab
wherein the data of a first data group has a different schema than the data of a second data group and **the data of the first data group is mapped differently than the data of the second data group** so that different lists of values are output for the corresponding different intermediate data, wherein the different schema and corresponding different intermediate data have a key in common; and	Data partitions can be processed by using different user-configured Mappers: **[4] HadoopMapReduce:** As key-value pairs are read from the RecordReader they are passed to the configured Mapper. The user supplied Mapper does whatever it wants with the input pair and calls OutputCollector.collect with key-value pairs of its own choosing. The output it generates must use one key class and one value class. This is because the Map output will be written into a SequenceFile which has per-file type information and all the records must have the same type (use subclassing if you want to output different data structures). The Map input and output key-value pairs are not necessarily related typewise or in cardinality.

Claim Chart – Claim 1

Claim Feature	Evidence from Charles Schwab
wherein **the data of a first data group has a different schema than the data of a second data group** and the data of the first data group is mapped differently than the data of the second data group so that different lists of values are output for the corresponding different intermediate data, wherein the **different schema** and corresponding different intermediate data have a key in common; and	The Schema class in Hive is defined for each data set. Thus, the data of a first data group can have a different schema than the data of a second data group. **[10] Class Schema:**

Modifier and Type	Class and Description
static class	Schema._Fields The set of fields this struct contains, along with convenience methods for finding and manipulating them.

Claim Chart – Claim 1

Claim Feature	Evidence from Charles Schwab
wherein the data of a first data group has a different schema than the data of a second data group and the data of the first data group is **mapped differently** than the data of the second data group so that different lists of values are output for the corresponding different **intermediate data**, wherein the different schema and corresponding different **intermediate data** have a key in common; and	The Mapper function maps the key-value pairs to form intermediate key-value pairs. Similarly, different Mappers generate different intermediate results: **[2] MapReduce Tutorial:** Mapper maps input key/value pairs to a set of intermediate key/value pairs. Maps are the individual tasks that transform input records into intermediate records. The transformed intermediate records do not need to be of the same type as the input records. A given input pair may map to zero or many output pairs.

Claim Chart – Claim 1

Claim Feature	Evidence from Charles Schwab

wherein the data of a first data group has a different schema than the data of a second data group and the data of the first data group is mapped differently than the data of the second data group so that different lists of values are output for the corresponding different intermediate data, wherein the different schema and corresponding different intermediate data **have a key in common**; and	A Hive JOIN query may be used to join two datasets that have a key in common:
	
	[12] LanguageManual Joins:
	
	Examples
	
	Some salient points to consider when writing join queries are as follows:
	
	• Complex join expressions are allowed e.g.
	
	`SELECT a.* FROM a JOIN b ON (a.id = b.id)`
	
	`SELECT a.* FROM a JOIN b ON (a.id = b.id AND a.department = b.department)`
	
	`SELECT a.* FROM a LEFT OUTER JOIN b ON (a.id <> b.id)`

Claim Chart – Claim 1

Claim Feature	Evidence from Charles Schwab
wherein the data of a first data group has a different schema than the data of a second data group and the data of the first data group is mapped differently than the data of the second data group so that different lists of values are output for the corresponding different intermediate data, wherein the different schema and corresponding different intermediate data have a key in common; and	Spark supports multiple types of structure data, each with its own schema. **[18] Learning Spark:**

Structured Data with Spark SQL

Spark SQL is a component added in Spark 1.0 that is quickly becoming Spark's preferred way to work with structured and semistructured data. By structured data, we mean data that has a *schema*—that is, a consistent set of fields across data records. Spark SQL supports multiple structured data sources as input, and because it understands their schema, it can efficiently read only the fields you require from these data sources. We will cover Spark SQL in more detail in Chapter 9, but for now, we show how to use it to load data from a few common sources.

Apache Hive

One common structured data source on Hadoop is Apache Hive. Hive can store tables in a variety of formats, from plain text to column-oriented formats, inside HDFS or other storage systems. Spark SQL can load any table supported by Hive.

JSON

If you have JSON data with a consistent schema across records, Spark SQL can infer their schema and load this data as rows as well, making it very simple to pull out the fields you need. To load JSON data, first create a `HiveContext` as when using Hive. (No installation of Hive is needed in this case, though—that is, you don't need a *hive-site.xml* file.) Then use the `HiveContext.jsonFile` method to get an RDD of Row objects for the whole file. Apart from using the whole Row object, you can also register this RDD as a table and select specific fields from it. For example, suppose that we had a JSON file containing tweets in the format shown in Example 5-33, one per line.

Claim Chart – Claim 1

Claim Feature	Evidence from Charles Schwab
wherein the data of a first data group has a different schema than the data of a second data group and the data of the first data group is mapped differently than the data of the second data group so that different lists of values are output for the corresponding different intermediate data, wherein the different schema and corresponding different intermediate data have a key in common; and	**[18] Learning Spark:**

File Formats

Spark makes it very simple to load and save data in a large number of file formats. Formats range from unstructured, like text, to semistructured, like JSON, to structured, like SequenceFiles (see Table 5-1). The input formats that Spark wraps all transparently handle compressed formats based on the file extension.

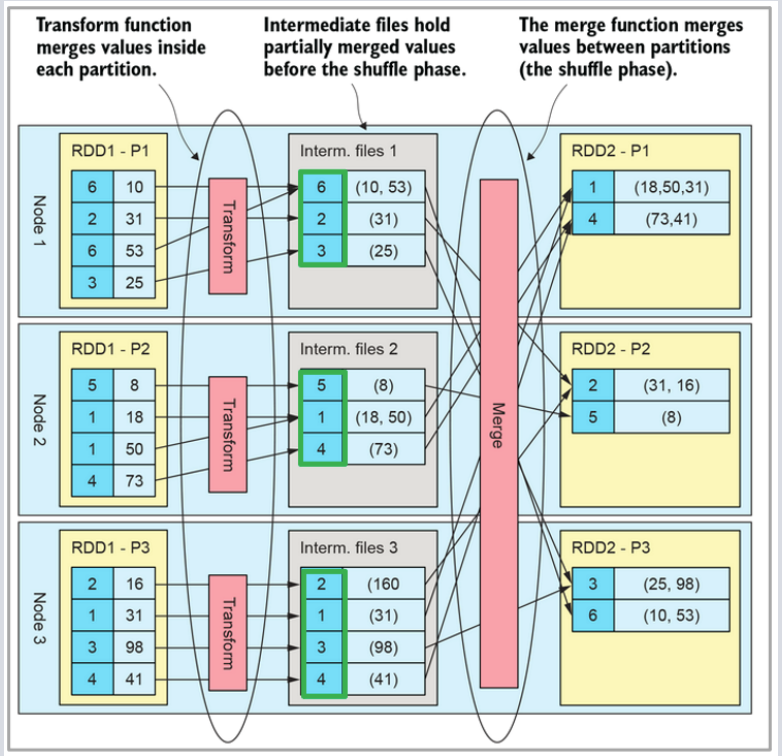
Table 5-1. Common supported file formats

Format name	Structured	Comments
Text files	No	Plain old text files. Records are assumed to be one per line.
JSON	Semi	Common text-based format, semistructured; most libraries require one record per line.
CSV	Yes	Very common text-based format, often used with spreadsheet applications.
SequenceFiles	Yes	A common Hadoop file format used for key/value data.
Protocol buffers	Yes	A fast, space-efficient multilanguage format.
Object files	Yes	Useful for saving data from a Spark job to be consumed by shared code. Breaks if you change your classes, as it relies on Java Serialization.

Claim Chart – Claim 1

Claim Feature	Evidence from Charles Schwab
wherein the data of a first data group has a different schema than the data of a second data group and **the data of the first data group is mapped differently than the data of the second data group so that different lists of values are output for the corresponding different intermediate data,** wherein the different schema and corresponding different intermediate data have a key in common; and	**[16] Apache Spark FAQ at 1:** How does Spark relate to Apache Hadoop? Spark is a fast and general processing engine compatible with Hadoop data. It can run in Hadoop clusters through YARN or Spark's standalone mode, and it can process data in HDFS, HBase, Cassandra, Hive, and any Hadoop InputFormat. It is designed to perform both batch processing (similar to MapReduce) and new workloads like streaming, interactive queries, and machine learning. **[23] LanguageManual Transform at 1:** Transform/Map-Reduce Syntax Users can also plug in their own custom mappers and reducers in the data stream by using features natively supported in the Hive language. e.g. in order to run a custom mapper script - map_script - and a custom reducer script - reduce_script - the user can issue the following command which uses the TRANSFORM clause to embed the mapper and the reducer scripts.

Claim Chart – Claim 1

Claim Feature	Evidence from Charles Schwab
wherein the data of a first data group has a different schema than the data of a second data group and the data of the first data group is mapped differently than the data of the second data group so that different lists of values are output for the corresponding different intermediate data, **wherein the different schema and corresponding different intermediate data have a key in common**; and	**[15] Spark in Action:**  Figure 4.2. A shuffle during the example `aggregateByKey` transformation on an RDD with three partitions. The transform function, passed to `aggregateByKey`, merges values in partitions. The merge function merges values between partitions during the shuffle phase. Intermediate files hold values merged per partition and are used during the shuffle phase.

Claim Chart – Claim 1

Claim Feature	Evidence from Charles Schwab
reducing the intermediate data for the data groups to at least one output data group, including processing the intermediate data for each data group in a manner that is defined to correspond to that data group, so as to result in a merging of the corresponding different intermediate data based on the key in common,	The method performed by the Charles Schwab system includes reducing the intermediate data for the data groups to at least one output data group, including processing the intermediate data for each data group in a manner that is defined to correspond to that data group, so as to result in a merging of the corresponding different intermediate data based on the key in common. The Reducer function reduces the intermediate values into a smaller set of values. This function operates in three phases: (1) Shuffle; (2) Sort; and (3) Reduce. The Shuffle and Sort phases operate on the output of various Mappers and group data before the final Reduce phase based on a common key. **[2] MapReduce Tutorial:** **Reducer** Reducer reduces a set of intermediate values which share a key to a smaller set of values. The number of reduces for the job is set by the user via Job.setNumReduceTasks(int). Overall, `Reducer` implementations are passed the `Job` for the job via the Job.setReducerClass(Class) method and can override it to initialize themselves. The framework then calls reduce(WritableComparable, Iterable<Writable>, Context) method for each `<key, (list of values)>` pair in the grouped inputs. Applications can then override the `cleanup(Context)` method to perform any required cleanup.

Claim Chart – Claim 1

Claim Feature	Evidence from Charles Schwab
reducing the intermediate data for the data groups to at least one output data group, including processing the intermediate data for each data group in a manner that is defined to correspond to that data group, so as to result in a merging of the corresponding different intermediate data based on the key in common,	[3] **Interface Reducer:**

Reduces a set of intermediate values which share a key to a smaller set of values.

The number of `Reducers` for the job is set by the user via `JobConf.setNumReduceTasks(int)`. `Reducer` implementations can access the `JobConf` for the job via the `JobConfigurable.configure(JobConf)` method and initialize themselves. Similarly they can use the `Closeable.close()` method for de-initialization.

`Reducer` has 3 primary phases:

1. **Shuffle**

   `Reducer` is input the grouped output of a `Mapper`. In the phase the framework, for each `Reducer`, fetches the relevant partition of the output of all the `Mappers`, via HTTP.

2. **Sort**

   The framework groups `Reducer` inputs by keys (since different `Mappers` may have output the same key) in this stage.

   The shuffle and sort phases occur simultaneously i.e. while outputs are being fetched they are merged.

Secondary Sort

If equivalence rules for keys while grouping the intermediates are different from those for grouping keys before reduction, then one may specify a `Comparator` via `JobConf.setOutputValueGroupingComparator(Class)`.Since `JobConf.setOutputKeyComparatorClass(Class)` can be used to control how intermediate keys are grouped, these can be used in conjunction to simulate *secondary sort on values*.

For example, say that you want to find duplicate web pages and tag them all with the url of the "best" known example. You would set up the job like:

- Map Input Key: url
- Map Input Value: document
- Map Output Key: document checksum, url pagerank
- Map Output Value: url
- Partitioner: by checksum
- OutputKeyComparator: by checksum and then decreasing pagerank
- OutputValueGroupingComparator: by checksum

3. **Reduce**

   In this phase the `reduce(Object, Iterator, OutputCollector, Reporter)` method is called for each <key, (list of values)> pair in the grouped inputs.

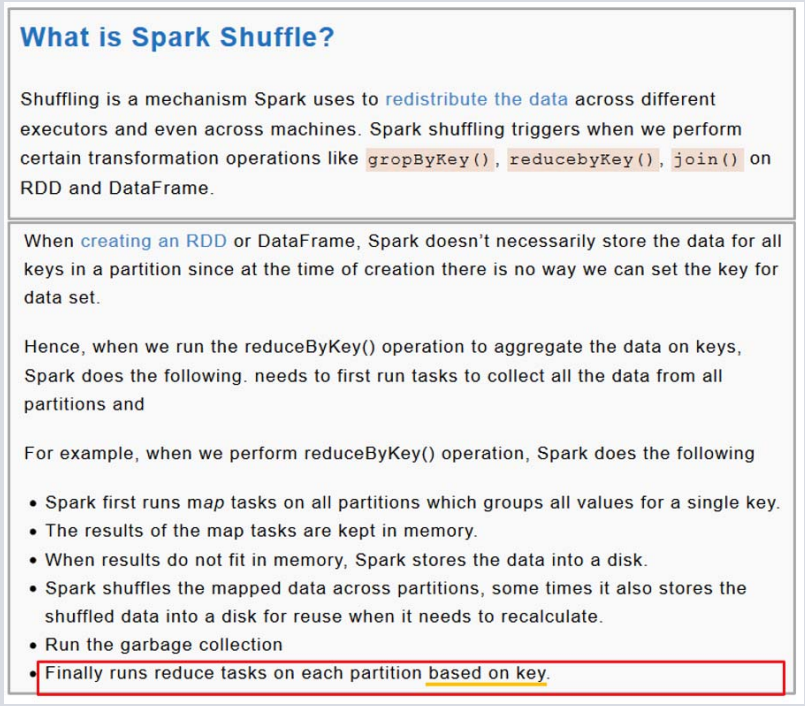
Claim Chart – Claim 1

Claim Feature	Evidence from Charles Schwab

reducing the intermediate data for the data groups to at least one output data group, including processing the intermediate data for each data group in a manner that is defined to correspond to that data group, so as to result in a **merging of the corresponding different intermediate data based on the key in common,**	The Reduce function operates on different sets of input data (intermediate data) that have a key in common. From this processing, the Reduce function populates an output file.
	
	[3] Interface Reducer:
	
	reduce
	`void reduce(K2 key, Iterator<V2> values, OutputCollector<K3,V3> output, Reporter reporter) throws IOException`
	Reduces values for a given key.
	The framework calls this method for each `<key, (list of values)>` pair in the grouped inputs. Output values must be of the same type as input values. Input keys must not be altered. The framework will **reuse** the key and value objects that are passed into the reduce, therefore the application should clone the objects they want to keep a copy of. In many cases, all values are combined into zero or one value.
	Output pairs are collected with calls to `OutputCollector.collect(Object,Object)`.
	Applications can use the `Reporter` provided to report progress or just indicate that they are alive. In scenarios where the application takes a significant amount of time to process individual key/value pairs, this is crucial since the framework might assume that the task has timed-out and kill that task. The other way of avoiding this is to set mapreduce.task.timeout to a high-enough value (or even zero for no time-outs).
	Parameters:
	`key` - the key.
	`values` - the list of values to reduce.
	`output` - to collect keys and combined values.
	`reporter` - facility to report progress.
	Throws:
	`IOException`

Claim Chart – Claim 1

Claim Feature	Evidence from Charles Schwab
reducing the intermediate data for the data groups to at least one output data group, including processing the intermediate data for each data group in a manner that is defined to correspond to that data group, so as to result in a merging of the corresponding different intermediate data based on the key in common,	In Spark, the intermediate data is reduced to an output data group. **[21] Spark SQL Shuffle Partitions:**



What is Spark Shuffle?

Shuffling is a mechanism Spark uses to redistribute the data across different executors and even across machines. Spark shuffling triggers when we perform certain transformation operations like `gropByKey()`, `reducebyKey()`, `join()` on RDD and DataFrame.

When creating an RDD or DataFrame, Spark doesn't necessarily store the data for all keys in a partition since at the time of creation there is no way we can set the key for data set.

Hence, when we run the reduceByKey() operation to aggregate the data on keys, Spark does the following. needs to first run tasks to collect all the data from all partitions and

For example, when we perform reduceByKey() operation, Spark does the following

- Spark first runs map tasks on all partitions which groups all values for a single key.
- The results of the map tasks are kept in memory.
- When results do not fit in memory, Spark stores the data into a disk.
- Spark shuffles the mapped data across partitions, some times it also stores the shuffled data into a disk for reuse when it needs to recalculate.
- Run the garbage collection
- Finally runs reduce tasks on each partition based on key.

Claim Chart – Claim 1

Claim Feature	Evidence from Charles Schwab
reducing the intermediate data for the data groups to at least one output data group, including processing the intermediate data for each data group in a manner that is defined to correspond to that data group, so as to result in a merging of the corresponding different intermediate data based on the key in common,	[15] **Spark in Action:**

4.2.2. Understanding and avoiding unnecessary shuffling

Physical movement of data between partitions is called *shuffling*. It occurs when data from multiple partitions needs to be combined in order to build partitions for a new RDD. When grouping elements by key, for example, Spark needs to examine all of the RDD's partitions, find elements with the same key, and then physically group them, thus forming new partitions.

To visualize what happens with partitions during a shuffle, we'll use the previous example with the `aggregateByKey` transformation (from section 4.1.2). The shuffle that occurs during that transformation is illustrated in figure 4.2.

The transform function puts all values of each key in a single partition (partitions P1 to P3) into a list. Spark then writes this data to intermediate files on each node. In the next phase, the merge function is called to merge lists from different partitions, but of the same key, into a single list for each key. The default partitioner (`HashPartitioner`) then kicks in and puts each key in its proper partition.

Tasks that immediately precede and follow the shuffle are called *map* and *reduce* tasks, respectively. The results of map tasks are written to intermediate files (often to the OS's filesystem cache only) and read by reduce tasks. In addition to being written to disk, the data is sent over the network, so it's important to try to minimize the number of shuffles during Spark jobs.

Claim Chart – Claim 1

Claim Feature	Evidence from Charles Schwab
reducing the intermediate data for the data groups to at least one output data group, including processing the intermediate data for each data group in a manner that is defined to correspond to that data group, so as to result in a merging of the corresponding different intermediate data based on the key in common,	**[15] Spark in Action:**

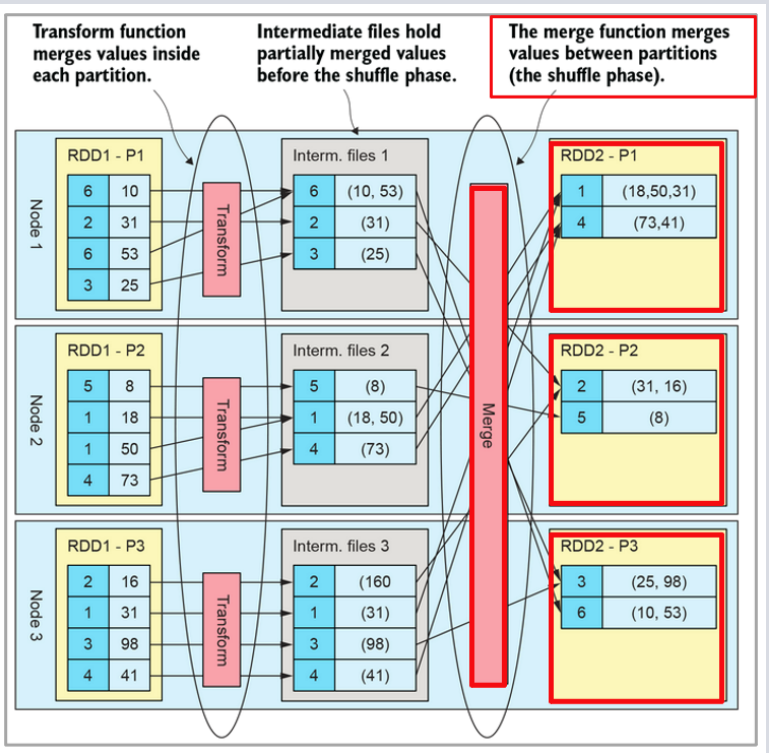


Figure 4.2. A shuffle during the example `aggregateByKey` transformation on an RDD with three partitions. The transform function, passed to `aggregateByKey`, merges values in partitions. The merge function merges values between partitions during the shuffle phase. Intermediate files hold values merged per partition and are used during the shuffle phase.

Claim Chart – Claim 1

Claim Feature	Evidence from Charles Schwab
wherein the mapping and reducing operations are performed by a **distributed system.**	The mapping and reducing operations are performed by the Charles Schwab system, which is a distributed system. **[1] An introduction to Apache Hadoop for big data:**

The Apache Hadoop framework is composed of the following modules

1. Hadoop Common: contains libraries and utilities needed by other Hadoop modules
2. Hadoop Distributed File System (HDFS): a distributed file-system that stores data on the commodity machines, providing very high aggregate bandwidth across the cluster
3. Hadoop YARN: a resource-management platform responsible for managing compute resources in clusters and using them for scheduling of users' applications
4. Hadoop MapReduce: a programming model for large scale data processing

All the modules in Hadoop are designed with a fundamental assumption that hardware failures (of individual machines, or racks of machines) are common and thus should be automatically handled in software by the framework. Apache Hadoop's MapReduce

END

nelson bumgardner albritton